Center for Reliable Computing

DYNAMIC ASSERTION TESTING OF FLIGHT CONTROL SOFTWARE

Dorothy M. Andrews, Aamer Mahmood and Edward J. McCluskey

CRC Technical Report No. 85-15

(CSL TN No. 85-274)

August 1985

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305 USA

Imprimatur: Mario Cortes and Aydin Ersoz

This work was supported in part by NASA-Ames Research Center under Grant No. NAG 2-246.

DYNAMIC ASSERTION TESTING OF FLIGHT CONTROL SOFTWARE

Dorothy M. Andrews, Aamer Mahmood, and Edward J. McCluskey

CRC Technical Report No. 85-15
(CSL TN No. 85-274)

August 1985

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Depts. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305 USA

ABSTRACT

This report describes an experiment in using assertions to dynamically test fault-tolerant flight software. The experiment showed that 87% of typical errors introduced into the program would be detected by assertions. Detailed analysis of the test data showed that the number of assertions needed to detect those errors could be reduced to a minimal set. The analysis also revealed that the most effective assertions tested program parameters that provided greater indirect (collateral) testing of other parameters.

Index Terms: Assertion Testing, Dynamic Software Testing, Data Dependency, Fault-Tolerant Software Testing, Flight Control Software.

TABLE OF CONTENTS

1 INTRODUCTION

In order to demonstrate the effectiveness of assertions in detecting errors in flight software, an experiment was conducted using Digital Flight Control System (DFCS) software as a test case [DFCR-96 80]. Flight software is real-time, has many logical variables, and uses fault-tolerant techniques, such as, voters and limiters built into the software. Assertions were written and embedded in the code by one person; then errors, chosen independently by another, were inserted (seeded) one at a time and the code was executed. The results from this experiment showed that 87% of the errors introduced into the DFCS program would be detected by assertions. Analysis of the research results demonstrated the following:

* Assertions are effective in detecting errors in digital flight control system software.

* The variables that are most dependent on other variables provide the greatest collateral (indirect) testing and, therefore, the assertions that test the "most dependent" variables are the most effective and detect the largest number of errors.

* Placement of assertions is an important factor in determining the effectiveness of an assertion, since those assertions placed in the procedures at the end of the calculations detected the most errors.

The fact that assertion testing proved to be effective for flight software has far reaching implications. The major one is that assertion testing can be used to eliminate errors at an earlier stage in the development cycle than before. Testing flight software has been extremely costly and time consuming because the elimination of errors has been done primarily by using simulators followed by actual flight testing. If the number of simulations and flights can be reduced because errors are detected sooner, there should be a considerable reduction in time and money spent on testing. In addition, because assertions have an excellent error detection rate, they can be used as a basis for implementing fault-tolerant techniques in flight software.

In this report, background information about assertion testing is presented first, then a description of the experiments, followed by a discussion of the research results and a conclusion.

2 ASSERTION TESTING

Assertion testing is a technique for dynamically testing software by embedding additional statements, called assertions, in the software. An assertion states a condition or specification in the form of a logical expression. During execution of the program, the assertion is evaluated as true or false. If it is false, the presence of an error is indicated. Notification of the error is most often made in an output message, such as, "Assertion in module <xxxx> at statement # <nn> is false."

Assertions can be written in the same language as the software, but they usually have a slightly different format (typically beginning with the word ASSERT) so they can be distinguished from the rest of the software. Before the program can be executed, the assertions must be translated into code that is acceptable by the compiler. This translation is done by a preprocessor, program analyzer, or a pre-compiler. Assertions are frequently made conditionally compilable, so they can be turned into comment statements and easily stripped out of the code after testing is finished.

Assertions may be inserted throughout the software, although sometimes they need only be added to certain strategic modules and still retain their effectiveness. An assertion can test the relationship between one or more variables, the range or limit of a variable, or

check the results of a numerical computation. Some examples of assertions are:

```
ASSERT (ABS (LAT_INN_CMD) > MAX_CPL)
ASSERT (ABS (K2 - 0.95133) > 0.0005)
ASSERT (ABS ((LAT_INN_CMD) - 0.5 * (RL5 + 0.753 * ROLL)) > 0.0001)
```

2.1 PROCEDURE FOR ASSERTION TESTING

Assertion testing differs from other forms of dynamic software testing (such as functional, random, or path testing) because assertions must be added to the code before it is executed. However, the generation of input test data can be the same as is used in any other testing procedure [Adrion 82], [Duran 84], [Gannon 79], [Howden 80], [Ntafos 85]. The procedure for assertion testing of software is as follows:

* Add assertions to the code - preferably this should be done during code implementation.

* Check correctness of the assertions (one way this may be done is by executing the code and, if any assertions are evaluated as false, determining whether the assertion is incorrect or an error is present).

* Generate test data automatically or by the usual testing methods and execute the program.

* When test runs are finished, assertions may be removed or

left in the code to provide on-line testing (especially in fault-tolerant applications).

Assertion testing has two distinct advantages over other testing methods: First, determining the correctness of the output is remarkably simplified because of the automatic notification of an error when an assertion is violated. Second, because of this reduction of time required for assessment of test results, the generation of a larger set of input data becomes possible and automation of the process of adaptively generating test data becomes easier to implement [Andrews 81,85], [Cooper 76].

2.2 FAULT-TOLERANT APPLICATIONS

Another important use of assertions is in building fault-tolerant systems [Randall 75], [Andrews 78,79]. A designer of a fault-tolerant system assumes that faults will occur and tries to prevent system failures by incorporating methods for error detection and correction during system operation. Assertions embedded in the software provide a convenient and effective way to implement on-line fault tolerance for hardware faults, as well as software errors. Assertions are used to detect the errors, and additional code (traditionally referred to as a recovery block) provides a way to handle the error. When an assertion is evaluated as false, control is transferred to the recovery block statements which are then executed. This technique allows implementation of a variety of responses to potentially critical problems.

3 RESEARCH EXPERIMENT

This section describes the flight control software used as a test case and the procedure followed in developing the assertions and generating the errors.

3.1 TEST CASE SOFTWARE

The software used as a test case was the autopilot code for a large, wide-bodied airplane. It is a good example of Digital Flight Control System software and is written in AED (Automated Engineer Design) [DFCR-96 80]. The software was written incrementally over the past decade and most of the "bugs" have been corrected. The code (which is installed on a flight simulator at the NASA AMES Research Center) is almost identical to that used in commercial planes at the present time.

The software is an integrated system that provides autopilot and flight director modes of operation for automatic and manual control of the plane during all phases of flight. The software is partitioned into five major categories: the first, of course, is control and navigation of the plane. In addition to this, are various supporting functions, namely, testing and voting, logic (engage and mode calculations), input/output (data handling, transmission, display, etc.), and the executive. Several procedures from the control and navigation category were used as the test case for the assertion testing experiment. These procedures use the selected heading and data from sensors and then calculate the commands to the ailerons (which cause the plane to change

direction). Figure 3.1 shows the relevant procedures and the flow of data.

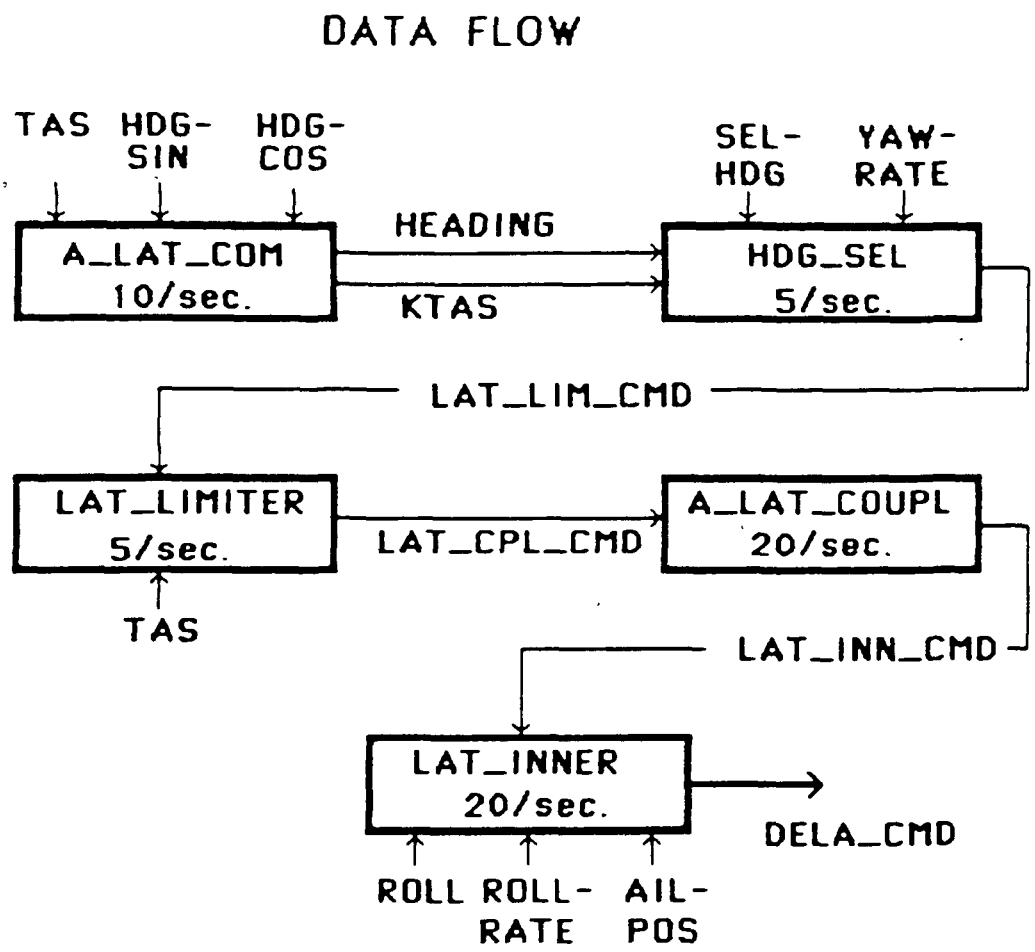

Fig 3.1 Flow of Data in Test Case Software

3.2 PROCEDURE

The original plan for the experiment was to add assertions, put in errors, and execute the autopilot code on the flight control computers installed at the Digital Flight Control Systems Verification Laboratory

at NASA-AMES [de Feo 82]. This was tried, but it proved to be more efficient to rewrite the program and execute it on another computer. The reasons for this are as follows: Developing assertions involves a certain amount of experimentation in order to refine them and measure the desired condition. In addition, the errors were to be seeded one at a time so it would be possible to determine whether or not a particular error had been detected. Each change in the code, for refinement of assertions or inserting an error, required recompilation of the entire program by an AED compiler which was on a Univac computer at a different location. Then the excutable code had to be downloaded into the flight computers on the pallet. It soon became apparent that the process of making changes to the code was so time consuming that very fews runs could be made in one day. For this reason, the code was rewritten in Pascal so it could be executed more efficiently on the DEC-20 at the Stanford University campus.

There were two other even more important reasons for moving the code to another computer. One was that introducing errors into the code often caused the flight computers installed on the pallet to "crash" (or not fly at all) because the effect of the error was so drastic. Consequently, the section of code containing assertions was never executed. The other reason was intrinsic to the nature of the flight computers which have a dual-dual redundancy architecture. Aberrations are corrected by voters and limiters built into the software [de Feo 82], so errors introduced in the software running on one channel would

be "corrected" by the voters or limiters before detection by an assertion.

In this experiment, the assertions were written by one person and the errors by another person. The reason for doing this was to maintain complete independence. Since existing documentation did not contain enough information to write assertions, the flight computers were run on the simulator in conjunction with a strip chart recording device to determine the normal values of the program variables. From this information, it was possible to write assertions for the set of modules to be tested. More detailed information may be found in the following: a description of the experiment to test flight software with assertions [Mahmood 84c]; suggestions for writing assertions in flight software gained from this experience [Mahmood 84a]. A combination of these two papers along with additional information was published as a technical report of the Center for Reliable Computing at Stanford University [Mahmood 84b].

The selection of errors was taken from two studies of errors made during implementation of flight control software [Hecht 82]; one was similar to the software used as a test case. Errors, chosen from four different classifications, were seeded one at a time in the software to determine the effectiveness of assertions in finding errors of different types. An effort was made to duplicate exactly the errors from the study whenever possible.

4 ANALYSIS OF RESEARCH RESULTS

This research study can be divided into three phases: the first was the original software testing on the flight simulators installed at the NASA-AMES Research Center; the second was conducting the tests on the DEC-20 at Stanford University; and the third was a detailed analysis of the factors affecting the effectiveness of the assertions themselves. This section describes the results from each phase.

4.1 FLIGHT SIMULATOR TESTING

The results of the first phase, although inconclusive since few tests were run on the flight simulator (because of the length of time required to run each test), contributed greatly to understanding the problems involved in testing real-time flight software. The first results clearly showed that testing a software system with built-in redundancy (that is, a fault tolerant system) is not possible using normal testing techniques. These results also indicated that the same problems encountered in testing fault tolerant hardware systems (fault masking, etc) exist for testing fault tolerant software systems and that "design for testability" features should be incorporated into fault-tolerant software design specifications.

When the software was executed on flight computers in a simulated real-time flight environment, the following characteristics of flight

software that contribute to the problems in testing fault-tolerant software were identified:

* USE OF LIMITERS In the autopilot code, there is frequent use of limiters which reset certain variables whose values are not within certain limits. This is done, not only to control possible errors, but also to keep the values of those variables within the limits of passenger comfort and within the stress limits of the airplane structure, etc. However, this use of limiters throughout the program interferes with detection of errors during testing because errors can be corrected by a limiter and therefore masked.

* USE OF VOTERS The values of input data, as well as the values of variables from computations, are continually voted upon. If one of the values does not agree with the others, the majority vote prevails. Therefore, errors can be masked and difficult to detect, since propagation of errors is halted.

* AUTOMATIC CHANNEL SYNCHRONIZATION The autopilot flight computers have a dual-dual redundancy architecture with automatic synchronization of the channels provided by the software. Under these conditions, assertions which monitor timing do not catch errors because timing problems are immediately corrected.

From these results, it was clear that it would be necessary to use different methods in order to test fault-tolerant flight software. Consequently, the software was tested as a single entity in a non-real-

time environment in subsequent runs. In other words, the redundancy and automatic channel synchronization had to be removed to be able to test flight software effectively without error masking. This same method, the disabling of redundancy, has been proposed for testing fault-tolerant hardware.

4.2 SIMULATION ON DEC-20 COMPUTER

When the flight software was executed on the DEC-20, eighty one errors were seeded (inserted) in the program one at a time to determine the effectiveness of assertions in finding errors of different types. The errors were from four different error classifications - data handling, logic, database, and computational. As Table 4.1 shows, nearly 70% of the errors were detected and, if the software had been fully asserted, nearly 90% of the seeded errors would have been detected. Assertions were put only in the part of the code executed during the heading select mode. The software was not fully asserted because some of the flight modes were not implemented on the flight simulator and, therefore, not enough information was available to simulate flight conditions for those modes correctly on the DEC-20. It was, however, possible to determine manually which of the errors could have been detected by assertions and which could not.

The errors (usually logic errors) that caused execution of the code for which assertions had not been written were not detected. The reason the remaining errors were not detected was frequently due to the fact that they had no effect on the computations. For example, Boolean

variables (having values of either 0 or 1) are typically assigned a value in flight software in statements such as, MODE = A or B or C and not D. Suppose A equals 1, then an error resulting in a change in value of B or C will have no effect on the outcome of this assignment statement and therefore would not be detected by an assertion. In another example, some errors changed the name of a Boolean variable into another. When the value of the variables was identical, such an error could not be detected.

Table 4.1 Types of Errors Detected by Assertions

ERROR TYPE	No. INSERTED	% ERRORS DETECTED	
		PARTIALLY ASSERTED	FULLY ASSERTED
DATA HANDLING	22	63.6	90.9
LOGIC	19	47.3	84.2
DATABASE	19	78.9	94.7
COMPUTATIONAL	21	76.1	80.9
TOTAL	81	66.6	87.6

4.3 OPTIMIZATION OF ASSERTION TESTING

After the experience in testing the flight control software, efforts were directed toward answering questions about both the qualitative and quantitative aspects of assertion testing. For example, how should assertions be written, what type of assertions are the most effective, where is the best placement for assertions, how many are needed, etc.

4.3.1 Writing the Assertions

From the difficulty experienced in this experiment in trying to write assertions with little knowledge of the program behavior and inadequate software specifications, it is obvious that assertions should be written in cooperation between a flight control analyst and the system designer or the programmer who is implementing the code. Some of the conditions that should be tested by assertions would only be known by flight specialists; and for that reason, it is imperative to have their help and guidance. The best time to add assertions is during the original coding, so the assertions will detect errors during module, as well as system integration testing.

4.3.2 Number of Assertions

The number of assertions depends on the phase of testing. When used for debugging, assertions should be embedded frequently throughout flight software code so they can point to the location of the errors. The suggested procedure is to seed the program with errors (as was done

in this experiment) and then retain a covering set of assertions, that is, the set detecting all seeded errors. Once the software is ready for testing in a flight simulation environment, then fewer assertions can be used so that memory space in the computer and execution time overhead are minimized.

When assertions are used for error detection in implementation of fault tolerance techniques, minimization of assertions (and the consequent overhead) is also important. The assumption would be that those assertions shown to be effective in error detection during the testing phase would be most able to detect intermittent and transient hardware faults, as well as any new software errors that might be introduced during maintainance.

Nineteen assertions were added to the software during the second phase of the experiment. Table 4.2 shows the number of errors detected by each of those assertions. Most errors were detected by more than one assertion. However, three errors were detected only by assertion number nineteen and a fourth error was detected only by assertion number seveteen. These two assertions constitute the set of "essential" or "critical" assertions if those four errors are to be detected. These two assertions also detected other errors. The remaining errors could be detected by either assertion number ten or fifteen. This means that out of all the assertions written, three assertions could be used to detect all the detectable errors. The implication of these results is that it may be possible to find a small subset of assertions capable of

detecting a large number of errors, so space and time overhead can be minimal. This result makes assertion testing even more attractive.

Table 4.2

Number of Errors Detected by Each Assertion

Assertion #	# Errors Detected
1	5
2	6
3	14
4	20
5	3
6	6
7	15
8	16
9	41
10	40
11	6
12	30
13	0
14	27
15	44
16	41
17	13
18	36
19	11

4.3.3 Placement of Assertions

The placement of the assertions is also dependent on the testing phase. During the early debugging phase, it is most desirable to have many assertions to check incoming data, outgoing commands, data storage and retrieval, and the results of computations. The analysis showed that the effective and essential assertions were in the last part of the asserted code (the procedures that calculate the final commands to the ailerons). This is not surprising since assertions placed earlier in the code would not catch errors introduced later on. Although the results of the first phase of the experiment showed that many of the seeded errors would be corrected by the voters built into the software, the results of the second phase demonstrated that assertions can detect those errors when the software system is tested as a single entity (with the redundancy disabled). In the testing phases where execution time and computer space are important, assertions should be placed in the procedures that calculate commands to the mechanical parts of a flight system.

4.3.4 Choice of Assertions

Assertions can be different types. They can measure the relationship between variables, check for maximum or minimum allowable values of a variable, or perform a numerical computation with a different algorithm to determine correctness of the calculation. Examples of each of these types of assertions were given in Sec. 2. Assertions also can have tests for more than one variable, and a factor

for time may also be included in the assertion. All types of assertions were written for this experiment and, interestly enough, none of these factors - type of assertion, inclusion of a factor for time, or checking multiple variables - seemed to affect the ability of the assertion to detect errors.

Further in-depth analysis did reveal a factor that appears to influence the effectiveness and criticality of an assertion. It seemed possible that testing certain variables might be more effective than testing others - depending on which variables provide greater collateral testing. One measure of collateral testing is the number of variables that are utilized in assigning a value to a variable. This number is refered to as the "data dependency" of that variable. The variables with the highest "data dependencies," therefore, would be expected to provide the greatest collateral testing of other variables.

This hypothesis was tested against the results from this experiment. A high correlation was found between the data dependency of the variables tested in an assertion and the effectiveness of the assertion. Those assertions with the highest accumulated data dependency factors (for the variables included in the assertion) proved to be the most effective in detecting errors. The difference in detection effectiveness was significant, since the assertions with high dependency factors detected ten times as many errors as the assertions with the lowest dependency factors. Not only did the most effective assertions have the highest data dependency factor, but the two

essential assertions also had very high dependency factors.

Therefore, to ensure that the greatest number of variables are directly or indirectly tested, the dependency factor for each variable should be calculated and the variables with the highest data dependency number should be included in the assertions. This relationship between assertion effectiveness and the data dependency factor of the variables being tested should be of considerable help in writing good assertions for flight software.

5 CONCLUSION

Initial test runs on flight computers installed on a flight simulator at NASA-AMES revealed major differences between real-time flight software and non-real-time software. More comprehensive testing done in the second testing phase indicated that, regardless of these differences, assertion testing is an effective method for detecting errors in flight control software. A major implication of this result is that assertion testing may be able to eliminate most errors at an earlier stage in the development cycle than before, thus reducing testing costs.

Therefore it is proposed that assertions be added to the software during implementation and that assertion testing be utilized from the beginning to shorten the testing cycle. Furthermore, in fault-tolerant computing applications, the suggested procedure is to retain the assertions during deployment and include additional code to provide error recovery. One of the conclusions reached as a result of this

experiment is that the number of assertions required to detect all possible detectable errors may be a small, minimal set - therefore making assertions a useful medium for providing fault tolerance in flight software.

According to these test results, effectiveness of an assertion was not affected by factors such as checking multiple variables, inclusion of a factor for time, or type of assertion. However, testing program parameters that provided the greatest collateral testing of other parameters improved the effectiveness of an assertion.

ACKNOWLEDGEMENT

This work was supported in part by NASA-AMES Research Center under Grant No. NAG 2-246. The authors would like to thank Roger Tapper of Rockwell International, and Dr. Dallas Denery, Jim Saito, and Doug Doane from NASA-AMES Research Center for their cooperation with this experiment. Appreciation is also due to the members of the Center for Reliable Computing at Stanford University for their suggestions.

REFERENCES

[Adrion 82] Adrion, W.R., M.A. Branstad and J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," ACM COMPUTING SURVEYS, Vol. 14, No. 2, pp. 159-192, June 1982.

[Andrews 78] Andrews, D.M., "Software Fault Tolerance Through Executable Assertions," Proc., 12TH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS AND COMPUTERS, Asilomar, CA., Nov. 1978.

[Andrews 79] Andrews, D.M., "Using Executable Assertions for Testing and Fault Tolerance," Proc., 1979 INT'L CONFERENCE ON FAULT-TOLERANT COMPUTING (FTCS-9), Madison, WI., June 20-22, 1979.

[Andrews 81] Andrews, D.M., and J. Benson, "An Automated Program Testing Methodology and Its Implementation," Proc., 5TH ANNUAL CONFERENCE ON SOFTWARE ENGINEERING, San Diego, CA., Mar. 9-12, 1981; Reprinted in Tutorial: SOFTWARE TESTING & VALIDATION TECHNIQUES, 2nd Edition, IEEE Computer Society Press, 1981.

[Andrews 85] Andrews, D. M., "Automation of Assertion Testing: Grid and Adaptive Techniques," Proceedings of the Hawaii International Conference on System Sciences (HICSS - 18), Honolulu, HA., Jan. 2-4, 1985.

[Cooper 76] Cooper, D.W., "Adaptive Testing," Proc., SECOND INT'L CONFERENCE ON SOFTWARE ENGINEERING, San Francisco, California, Oct. 13-15, 1976.

[de Feo 82] de Feo, P., D. Doane, and J. Saito, "An Integrated User-Oriented Laboratory for Verification of Digital Flight Control Systems - Features and Capabilities," NASA Technical Memorandum 84276, Ames Research Center, Moffett Field, CA., 94035, Aug. 1982.

[DFCR-96 80] L-1011 DAFCS Software Description, DFCR-96R1, L-1011 Digital Flight Control System Report, Collins Avionics Division, Rockwell International, 1980.

[Duran 84] Duran, J.W., "An Evaluation of Random Testing," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-10, No. 4, July 1984.

[Gannon 79] Gannon, C., "Error Detection Using Path Testing and Static Analysis," COMPUTER, Vol. 12, Aug. 1979.

[Hecht 83] Hecht, H. and M. Hecht, "Trends in Software Reliability of Digital Flight Control Systems," NASA Technical Report No. 166456, Ames Research Center, Moffet Field, CA., 94035, Apr. 1983.

[Howden 80] Howden, W.E., "Functional Program Testing," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-6, No. 2, Mar. 1980.

15

[Mahmood 84a] Mahmood, A., D.M. Andrews and E.J. McCluskey, "Writing Executable Assertions to Test Flight Software," Proc., 18TH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS AND COMPUTERS, Pacific Grove, CA., Nov. 4-7, 1984.

[Mahmood 84b] Mahmood, A., D. M. Andrews and E. J. McCluskey, "Executable Assertions and Flight Software," CRC Technical Report No. 84-16, CSL TN No. 84-258, Nov. 1984.

[Mahmood 84c] Mahmood, A., D.M. Andrews and E.J. McCluskey, "Executable Assertions and Flight Software," Proc., 1984 IEEE/AIAA 6TH DIGITAL AVIONICS SYSTEMS CONFERENCE, Baltimore, Maryland, Dec. 3-6, 1984.

[Ntafos 85] Ntafos, Simeon C., " An Investigation of Stopping Rules for Random Testing," Proceedings of the Hawaii International Conference on System Sciences (HICSS - 18), Honolulu, Hawaii, Jan. 2-4, 1985.

[Randall 75] Randall, B., "System Structure for Software Fault Tolerance," PROC. INT'L CONFERENCE ON RELIABLE SOFTWARE, Los Angeles, California, Apr. 1975.